

**System and Method for Processor Thread
Acting as a System Service Processor**

EL977166465US

BACKGROUND OF THE INVENTION

1. Technical Field

5 The present invention relates in general to a system and method for using a processor thread to act as a service processor. More particularly, the present invention relates to a system and method for initiating a service thread which monitors service events and also invokes a
10 second thread to perform operational tasks.

2. Description of the Related Art

 A large computer system executes an enormous amount of tasks. These large computer systems typically have a dedicated processor which monitors the computer system to
15 ensure that the computer system is functioning properly. This dedicated processor, called a service processor, is responsible for monitoring the computer system and detecting potential problems and safeguarding the computer system. For example, a service processor may perform tasks
20 such as an emergency shutdown in an orderly manner when the service processor detects a problem.

 A service processor monitors physical states such as temperatures, currents, and voltage levels of computer system hardware. When a service processor detects an
25 excessive measurement, such as excessive heat, the service processor may shut down the computer system in a manner that does not lose operational data. In addition, a service processor may monitor events such as corrected and

uncorrectable memory and cache errors. For example, when a certain error rate is exceeded or a trend exists that indicates a potential future problem, the system administrator may be notified.

5 A service processor performs vital functions that increase the robustness of a computer system. A challenge found, however, is that a service processor is not practical for use in consumer electronic applications. A service processor generates heat and consumes power.
10 However, consumer electronic manufactures attempt to reduce heat and power in the electronic devices they produce.

 In addition, a service processor requires board space. Consumer electronic manufacturers attempt to reduce the size of electronic equipment, and adding an additional
15 processor conflicts with their efforts.

 Furthermore, consumer electronic equipment tends to be low cost. Adding a service processor to a bill of materials increases the cost of a consumer electronic device which may make it uncompetitive.

20 What is needed, therefore, is a system and method for performing service processor tasks without adding an additional processor to perform such tasks.

SUMMARY

It has been discovered that the aforementioned challenges are resolved by initiating a service thread to perform service processor tasks and using the service
5 thread to invoke an operational thread to run an operating system and execute programs. When the service thread detects a problem with one of the service events or the operational thread, the service thread may terminate the operational thread and/or save operational data for future
10 analysis.

When a computer system boots-up, a processor starts a service thread which is responsible for performing tasks similar to what a service processor performs in a multi-processor architecture. For example, the service thread
15 performs service checks on system events such as ECC error checks, hardware error log checks, heat sensor checks, and fan activity checks. The service thread uses service code to identify and monitor service events and accesses its own register set during the service thread's operation.

20 Once the service thread is functioning, the service thread invokes a second thread to perform operational tasks (i.e. an operational thread), such as running an operating system and a kernel. Once invoked, the operational thread loads a primary operating system from system memory and
25 runs various application programs. For example, the operational thread may run a gaming application using the primary operating system. In this example, the service thread monitors service events and the operational thread while the operational thread executes the gaming

application. Like the service thread, the operational thread accesses its own register set to run applications.

During execution, the service thread receives service event values from service events and stores the values in a storage area. For example, the service thread may receive a temperature value from a heat sensor and store the value the storage area. When the service thread detects a service error with one of the service events (i.e. excessive current), the service thread attempts to correct the problem by adjusting one or more service tolerances. If the problem is not correctable, or if the service thread determines that the operational thread is not functioning, the service thread terminates the operational thread and stores operational data in a storage area for future analysis.

The foregoing is a summary and thus contains, by necessity, simplifications, generalizations, and omissions of detail; consequently, those skilled in the art will appreciate that the summary is illustrative only and is not intended to be in any way limiting. Other aspects, inventive features, and advantages of the present invention, as defined solely by the claims, will become apparent in the non-limiting detailed description set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the
5 accompanying drawings. The use of the same reference symbols in different drawings indicates similar or identical items.

Figure 1 is a diagram showing a first thread acting as a service processor and a second thread performing
10 operational tasks;

Figure 2 is a service thread invoking an operational thread and monitoring service events;

Figure 3 is a flowchart showing steps taken in a service processor thread performing system checks on a
15 computer system;

Figure 4 is a flowchart showing steps taken in a service thread analyzing an operating system thread's functionality;

Figure 5 is a diagram showing steps taken in a
20 debugger thread monitoring an operational thread;

Figure 6 is a flowchart showing steps taken in a debugger thread monitoring an operational thread;

Figure 7 is a flowchart showing steps taken in a debugger thread analyzing an operational thread's failure;
25 and

Figure 8 is a block diagram illustrating a processing element having a main processor and a plurality of secondary processors sharing a system memory.

DETAILED DESCRIPTION

The following is intended to provide a detailed description of an example of the invention and should not be taken to be limiting of the invention itself. Rather,
5 any number of variations may fall within the scope of the invention which is defined in the claims following the description.

Figure 1 is a diagram showing a first thread acting as a service processor and a second thread performing operational tasks. Computer system **100** includes processor
10 **110**. Processor **110** invokes one or more threads which are responsible for particular tasks. The example shown in **Figure 1** shows service thread **120** performing service processor tasks and operational thread **170** performing
15 operational tasks, such as executing an operating system.

When processor **110** boots-up, processor **110** starts service thread **120** which is responsible for performing tasks similar to what a service processor performs in a multi-processor architecture. For example, service thread
20 **120** performs service checks on system events (e.g. service events **125**) such as Error Checking and Correcting Memory (ECC) checks, hardware error log checks, heat sensor checks, and fan activity checks. Service thread **120** uses service code **130** to identify and monitor service events
25 **125**. Service thread **120** uses its own register set, such as service register set **140**, during service thread **120**'s operation.

Once service thread **120** is functioning, service thread **120** invokes a second thread to perform operational tasks

(e.g. operational thread **170**), such as running an operating system and a kernel. Once invoked, operational thread **170** loads primary operating system **180** from memory store **115** and runs application programs that computer system **100** require. For example, operational thread **170** may run a gaming application using primary operating system **180**. In this example, service thread **120** monitors system events **125** as well as operational thread **170** while operational thread **170** executes the gaming application. Memory store **115** may be stored on a nonvolatile storage area, such as a computer hard drive. Like service thread **120**, operational thread **170** uses its own register set, such as primary registers **190**.

During execution, service thread **120** receives service event values from service events **125** and stores them in event store **135**. For example, service thread **120** may receive a temperature value from a heat sensor and store the value in event store **135**. Event store **135** may be stored on a volatile or nonvolatile storage area, such as computer memory. When service thread **120** detects a service error with one of the service events, service thread **120** attempts to correct the problem by adjusting service tolerances (see **Figure 3** and corresponding text for further details service tolerance adjustments).

When service thread **120** detects that operational thread **170** is not functioning, service thread **120** backs up operational thread **170**'s data and/or properties and stores them in backup store **145**. For example, service thread **120** may retrieve register values from primary registers **190** and store the values in backup store **145** for later analysis. For example, an operational thread that is not functioning

may be reset whereby code at its entry point may be used to dump register information to a service thread (see **Figure 4** and corresponding text for further details regarding operational data analysis.

5 **Figure 2** is a service thread invoking an operational thread and monitoring service events. A computer system boots up whereupon service thread processing commences at **200**. The service thread loads firmware at step **205**. In one embodiment, the service thread loads an operating
10 system to assist in service event monitoring.

The service thread invokes the operational thread at step **210**. The operational thread is a processing thread separate than the service thread, but executed on the same processor as the service thread. Operational thread
15 processing commences at **215**. The operational thread is responsible for running a primary operating system within the computer system, such as Linux. The operational thread loads an operating system from memory store **115** at step **220**. Memory store **115** is the same as that shown in **Figure**
20 **1**. The operational thread begins executing the primary operating system at step **235**.

The service thread loads service code from memory store **115** at step **230**. Service code includes code to check particular service events and also includes expected values
25 of the service events. For example, one of the service events may be the proper operation of a fan. In this example, the service thread monitors the fan operation and validates that the fan is activated when a computer system reaches a particular temperature.

The service thread performs service checks to the computer system, such as checking service event values that correspond to fan **255** and sensor **260**. Using the example described above, sensor **260** may provide a temperature
5 reading that corresponds to the temperature of the computer system, and the service thread validates that the fan is active when the computer system reaches a particular temperature (pre-defined process block **250**, see **Figure 3** and corresponding text for further details).

10 A determination is made as to whether the computer system passes service checks (decision **265**). For example, the service thread may log physical parameters (i.e. temperature, voltage, etc.) when an error occurs. In this example, over a period of time, a service thread may detect
15 a correlation between one of the physical parameters and a system error in which the service thread identifies non-recoverable software errors in the OS and locate non-recoverable memory or cache errors. If the computer system passes service checks, decision **265** branches to "Yes"
20 branch **269** whereupon the service thread monitors the operational thread's functionality (pre-defined process block **270**, see **Figure 4** and corresponding text for further details). A determination is made as to whether to terminate operational thread processing (decision **280**).
25 For example, the service thread may detect a problem with a particular service event other operational thread itself. If the service thread should not terminate the operational thread, decision **280** branches to "Yes" branch **282** which loops back to continue service processing. This looping
30 continues until the service thread should terminate the operational thread, at which point decision **280** branches to

"No" branch **288** whereupon the service thread terminates the operational thread at step **285**, and service thread processing ends at **290**.

5 A determination is made as to whether the service thread wishes to terminate the operational thread (decision **240**). If the operational thread should continue, decision **240** branches to "Yes" branch **242** which loops back to continue execution. This looping continues until the operational thread receives a termination request from the
10 service thread, at which point decision **240** branches to "No" branch **244** whereupon operational thread processing ends at **245**.

Figure 3 is a flowchart showing steps taken in a service processor thread performing system checks on a
15 computer system. Service thread processing commences at **300**, whereupon the service thread performs system checks on service events **125** (step **310**). Service events **125** is the same as that shown in **Figure 1** and includes events such as ECC error checks, hardware error log checks, heat sensor
20 checks, and fan activation checks. For example, a service thread may use a polling loop to read registers at a fixed rate and receive interrupts for critical or non-recoverable errors.

Processing logs service event values that correspond
25 to the service events in event store **135** (step **320**). For example, the service thread may retrieve a temperature value from a heat sensor, and determine whether the fan should be active at the measured temperature. In this example, the service thread logs the temperature value and
30 the state (i.e. on or off) of the fan. Event store **135** is the same as that shown in **Figure 1** and may be stored on a

nonvolatile or volatile storage area, such as computer memory.

5 A determination is made as to whether service events **125** passed system checks (decision **330**). Using the example described above, processing checks the fan's state to verify that the state is correct based upon the heat sensor value. If each of the system checks passes, decision **330** branches to "Yes" branch **332** whereupon processing returns a "Pass" value at **335**.

10 On the other hand, if one of the system checks fails, decision **330** branches to "No" branch **338** whereupon processing analyzes the service event values stored in event store **135** (step **340**). For example, if a fan was not
15 turned on when it should have been at a particular temperature, processing may analyze a calibration factor that is used to translate a heat sensor's value to a particular temperature. In this example, the service processor may analyze the computational load of a processor to identify the amount of heat the processor should be
20 producing. A determination is made as to whether the system event failure is correctable (decision **350**). If processing determines that the system event failure is not correctable, decision **350** branches to "No" branch **358** whereupon the service processor backs up operational thread
25 **170** and stores the operational data in backup store **145** (step **390**). The backup data may include operational thread **170**'s register values and interrupt values. Operational thread **170** and backup store **145** are the same as that shown in **Figure 1**. Processing returns a "Fail" at **395**.

30 On the other hand, if the system event failure is correctable, decision **350** branches to "Yes" branch **352**

whereupon processing adjusts event tolerances located in tolerances store 365 (step 360). Using the example described above, processing may increase a calibration factor corresponding to the heat sensor in order to
5 activate the fan at a lower temperature.

Processing performs system checks on system events 125 with the adjusted event tolerances at step 370. A determination is made as to whether the adjusted event tolerances corrected the service event failure (decision
10 380). Using the example described above, processing determines whether the fan's activation in response to a heat value is correct. If the system checks pass, decision 380 branches to "Yes" branch 388 whereupon processing returns a "Pass" at 385. On the other hand, if processing
15 determines that the event failure is not correctable, decision 380 branches to "No" branch 382 whereupon the service processor backs up operational thread 170 and stores the backup data in backup store 145 (step 390), and returns a "Fail" at 395.

20 **Figure 4** is a flowchart showing steps taken in a service thread analyzing an operating system thread's functionality. Thread monitoring processing commences at 400, whereupon the service thread polls operational thread 170 at step 410. For example, the service thread uses
25 shared memory to communicate with an operational thread and expects a response within a particular amount of time. A determination is made as to whether operational thread 170 is functioning. For example, operational thread 170 may have halted due to a particular interrupt value and is not
30 responding to the service thread's polls. If operational

thread **170** is functioning, decision **420** branches to "Yes" branch **422** whereupon processing returns a "Pass" at **425**.

On the other hand, if operational thread **170** is not functioning, decision **420** branches to "No" branch **428** whereupon the service thread backs up operational thread **170** and stores the backed up data in backup store **145**. The backup data may include operational thread **170**'s register values and/or interrupt values. Processing analyzes the failure at step **440** by identifying the failure mode and reviewing particular bits. For example, processing may receive register values corresponding to the operational thread whereby the service thread identifies a memory location of the occurring error and a corresponding instruction. In this example, the service thread examines the registers to determine if an uncorrectable memory error occurred and may deem the memory location has having an uncorrectable error when it is read.

A determination is made as to whether the failure mode was due to event tolerances (decision **450**). For example, the service thread may log physical parameters (i.e. temperature, voltage, etc.) when an error occurs. In this example, over a period of time, a service thread may detect a correlation between one of the physical parameters and a system error in which the service thread identifies non-recoverable software errors in the OS and locate non-recoverable memory or cache errors. If processing determines that operational thread **170**'s failure is not due to event tolerances, decision **450** branches to "No" branch **452** whereupon processing returns a "Fail" at **455**.

On the other hand, if operational thread **170** failed due to an event tolerance, decision **450** branches to "Yes"

branch **458** whereupon processing adjusts event tolerances in event tolerances store **365** (step **460**). Using the example described above, processing may adjust a temperature calibration value which turns on a computer system's fan at a lower temperature. Tolerances store **365** is the same as that shown in **Figure 3**.

Processing resets operational thread **170** at step **470**. Once reset, processing polls operational thread **170** at step **480**. In one embodiment, processing waits a particular time before re-polling operational thread **170** in order for operational thread **170** to reach a particular area in code where operational thread **170** previously failed.

A determination is made as to whether operational thread **170** is functioning (decision **485**). If operational thread is not functioning, decision **485** branches to "No" branch **487** whereupon processing returns a "Fail" at **490**. On the other hand, if operational thread **170** is functioning, decision **485** branches to "Yes" branch **489** whereupon processing returns a "Pass" at **495**.

Figure 5 is a diagram showing steps taken in a debugger thread monitoring an operational thread. Processor **110** invokes threads which are responsible for particular tasks. The example shown in **Figure 5** shows debugger thread **500** and operational thread **170** running on processor **110**. Processor **110**, operational thread **170**, primary operating system **180**, and primary registers **190** are the same as that shown in **Figure 1**.

When processor **110** boots-up, processor **110** starts debugger thread **500**. Developer **550** uses debugger thread **500** to debug code running on operational thread **170**, such

as primary operating system **180**. For example, operational thread **170** may be executing a new, fragile operating system, such as an untested gaming operating system, and debugger thread **500** may be used to debug the gaming
5 operating system when it fails. Debugger thread **500** loads its own operating system, such as debugger operating system **510**. Debugger operating system **510** may be a more stable operating system than primary operating system **180** because developer **550** requires debugger thread **500** to be robust in
10 order to debug operational thread **170**. Debugger operating system **510** uses its own registers for storing and manipulating information, such as debugger registers **520**.

Once debugger thread **500** is running, debugger thread **500** invokes a second thread (e.g. operational thread **170**)
15 to perform operational tasks, such as running an operating system and running a kernel. Once invoked, operational thread **170** loads primary operating system **180** and may run application programs as well. For example, primary operating system **180** may be a new gaming operating system
20 which runs particular graphic applications. Like debugger thread **500**, operational thread **170** has its own register set, such as primary registers **190**.

Debugger thread **500** monitors the performance and state of operational thread **170**. When debugger thread **500**
25 detects that operational thread **170** has failed or is stuck in an infinite loop, debugger thread **500** notifies developer **550** through debugger API **540**. Once notified, developer **550** sends requests to debugger thread to extract operational data from operational thread **170** to assist developer **550** in
30 the debugging process, such as register values, code line failure values, interrupts, and memory values. For

example, developer **550** may request debugger thread **500** to extract register values from primary registers **190** and store the values in debug store **530** for developer **550** to analyze (see **Figure 7** and corresponding text for further
5 details regarding operational data analysis). Debug store **530** may be stored on a nonvolatile or volatile storage area, such as computer memory.

Figure 6 is a flowchart showing steps taken in a debugger thread monitoring an operational thread. Debugger
10 thread processing commences at **600**, whereupon processing loads firmware at step **605**. Processing then loads an operating system for the debugger thread to use from debug store **530** (step **610**). The operating system is typically a stable operating system, such as Linux, which allows the
15 debugger thread to be robust during debugging activities. Debug store **530** is the same as that shown in **Figure 5**.

The debugger thread invokes an operational thread at step **615**, whereupon operational thread processing commences at **620**. The operational thread loads its primary operating
20 system from primary store **630** at step **625**. The primary operating system is code that the debugger thread monitors and may not be as robust as the debugger threads operating system. For example, the primary operating system may be a new, untested, gaming operating system. Primary store **630**
25 may be stored on a nonvolatile or volatile storage area, such as computer memory.

The operational thread executes the primary operating system code at step **635**. A determination is made as to whether the primary operating system code has failed
30 (decision **640**). For example, the primary operating system may have ventured into an infinite loop. If the primary

operating system has not failed, decision **640** branches to "No" branch **642** which loops back and continues to execute the primary operating system. This looping continues until the primary operating system fails, at which point decision
5 **640** branches to "Yes" branch **644** whereupon operational thread processing ends at **645**.

While the operational thread is executing the primary operating system, the debugger thread monitors the operational thread and may provide data to developer **550**
10 (step **650**). Developer **550** is the same as that shown in **Figure 5**. A determination is made as to whether the operational thread failed (decision **660**). If the operational thread did not fail, decision **660** branches to "No" branch **662** which loops back to continue to monitor the
15 operational thread. This looping continues until the operational thread fails, at which point decision **660** branches to "Yes" branch **664** whereupon the debugger thread debugs the operational thread (pre-defined process block **670**, see **Figure 7** and corresponding text for further
20 details).

A determination is made as to whether to reset the operational thread (decision **680**). For example, developer **655** may have fixed a code segment and wishes to re-run the operational thread. If the operational thread should be
25 reset, decision **680** branches to "Yes" branch **682** which loops back to reset and monitor the operational thread. This looping continues until the debugger thread should not reset the operational thread, at which point decision **680** branches to "No" branch **688** whereupon debugger thread
30 processing ends at **690**.

Figure 7 is a flowchart showing steps taken in a debugger thread analyzing an operational thread's failure. Debugger thread processing commences at **700**, whereupon the debugger thread notifies developer **550** of the operational thread failure (step **710**). For example, the debugger thread may send a message to developer **550** that the operational thread failed. Developer **550** is the same as that shown in **Figure 5**. The debugger thread receives a request from developer **550** at step **720**.

A determination is made as to whether developer **550** wishes to capture the register state of operational thread **170** (decision **730**). Operational thread **170** is the same as that shown in **Figure 1**. If developer **550** wishes to capture the register state of operational thread **170**, decision **730** branches to "Yes" branch **732** whereupon the debugger thread retrieves register values from operational thread **170** and stores the values in debug store **530**. Debug store **530** is the same as that shown in **Figure 5**. On the other hand, if developer **550** does not wish for the debugger thread to capture register values, decision **730** branches to "No" branch **738**.

A determination is made as to whether developer **550** wishes to capture interrupts corresponding to operational thread **170** (decision **740**). If developer **550** wishes to capture interrupts corresponding to operational thread **170**, decision **740** branches to "Yes" branch **742** whereupon the debugger thread retrieves interrupt values from operational thread **170** and stores the values in debug store **530**. On the other hand, if developer **550** does not wish for the debugger thread to capture interrupt values, decision **740** branches to "No" branch **748**.

A determination is made as to whether developer 550 wishes to capture a failed code line corresponding to operational thread 170 (decision 750). For example, developer 550 may wish to identify the code line location at which the operational thread failed. If developer 550 wishes to capture a failed code line corresponding to operational thread 170, decision 750 branches to "Yes" branch 752 whereupon the debugger thread retrieves a failed code line value from operational thread 170 and stores the values in debug store 530. On the other hand, if developer 550 does not wish for the debugger thread to capture a failed code line, decision 750 branches to "No" branch 758.

A determination is made as to whether developer 550 wishes to acquire more operational data from operational thread 170 (decision 760). If developer 550 wishes to receive more requests, decision 760 branches to "Yes" branch 762 which loops back to receive and process more requests. This looping continues until developer 550 does not wish to send more requests, at which point decision 680 branches to "No" branch 768. Developer 550 analyzes operational data located in debug store 530 at step 770. After analysis, developer 550 fixes the code and reloads the code in primary operating system store 630 at step 780, and debugger thread processing returns at 790. Primary operating system store 630 is the same as that shown in **Figure 6**. As one skilled in the art can appreciate, a debugger thread may perform other debug tasks besides what is shown in **Figure 7**, such as stack back tracing and viewing memory locations corresponding to the operational thread.

Figure 8 is a block diagram illustrating a processing element having a main processor and a plurality of secondary processors sharing a system memory. Processor Element (PE) **805** includes processing unit (PU) **810**, which, in one embodiment, acts as the main processor and runs an operating system. Processing unit **810** may be, for example, a Power PC core executing a Linux operating system. PE **805** also includes a plurality of synergistic processing complex's (SPCs) such as SPCs **845**, **865**, and **885**. The SPCs include synergistic processing units (SPUs) that act as secondary processing units to PU **810**, a memory storage unit, and local storage. For example, SPC **845** includes SPU **860**, MMU **855**, and local storage **859**; SPC **865** includes SPU **870**, MMU **875**, and local storage **879**; and SPC **885** includes SPU **890**, MMU **895**, and local storage **899**.

Each SPC may be configured to perform a different task, and accordingly, in one embodiment, each SPC may be accessed using different instruction sets. If PE **805** is being used in a wireless communications system, for example, each SPC may be responsible for separate processing tasks, such as modulation, chip rate processing, encoding, network interfacing, etc. In another embodiment, the SPCs may have identical instruction sets and may be used in parallel with each other to perform operations benefiting from parallel processing.

PE **805** may also include level 2 cache, such as L2 cache **815**, for the use of PU **810**. In addition, PE **805** includes system memory **820**, which is shared between PU **810** and the SPUs. System memory **820** may store, for example, an image of the running operating system (which may include the kernel),

device drivers, I/O configuration, etc., executing applications, as well as other data. System memory **820** includes the local storage units of one or more of the SPCs, which are mapped to a region of system memory **820**. For
5 example, local storage **859** may be mapped to mapped region **835**, local storage **879** may be mapped to mapped region **840**, and local storage **899** may be mapped to mapped region **842**. PU **810** and the SPCs communicate with each other and system memory **820** through bus **817** that is configured to pass data
10 between these devices.

The MMUs are responsible for transferring data between an SPU's local store and the system memory. In one embodiment, an MMU includes a direct memory access (DMA) controller configured to perform this function. PU **810** may
15 program the MMUs to control which memory regions are available to each of the MMUs. By changing the mapping available to each of the MMUs, the PU may control which SPU has access to which region of system memory **820**. In this manner, the PU may, for example, designate regions of the
20 system memory as private for the exclusive use of a particular SPU. In one embodiment, the SPUs' local stores may be accessed by PU **810** as well as by the other SPUs using the memory map. In one embodiment, PU **810** manages the memory map for the common system memory **820** for all the
25 SPUs. The memory map table may include PU **810**'s L2 Cache **815**, system memory **820**, as well as the SPUs' shared local stores.

In one embodiment, the SPUs process data under the control of PU **810**. The SPUs may be, for example, digital
30 signal processing cores, microprocessor cores, micro

controller cores, etc., or a combination of the above cores. Each one of the local stores is a storage area associated with a particular SPU. In one embodiment, each SPU can configure its local store as a private storage area, a
5 shared storage area, or an SPU may configure its local store as a partly private and partly shared storage.

For example, if an SPU requires a substantial amount of local memory, the SPU may allocate 100% of its local store to private memory accessible only by that SPU. If,
10 on the other hand, an SPU requires a minimal amount of local memory, the SPU may allocate 10% of its local store to private memory and the remaining 90% to shared memory. The shared memory is accessible by PU 810 and by the other SPUs. An SPU may reserve part of its local store in order
15 for the SPU to have fast, guaranteed memory access when performing tasks that require such fast access. The SPU may also reserve some of its local store as private when processing sensitive data, as is the case, for example, when the SPU is performing encryption/decryption.

20 One of the preferred implementations of the invention is an application, namely, a set of instructions (program code) in a code module which may, for example, be resident in the random access memory of the computer. Until
25 required by the computer, the set of instructions may be stored in another computer memory, for example, on a hard disk drive, or in removable storage such as an optical disk (for eventual use in a CD ROM) or floppy disk (for eventual use in a floppy disk drive), or downloaded via the Internet or other computer network. Thus, the present invention may
30 be implemented as a computer program product for use in a

computer. In addition, although the various methods described are conveniently implemented in a general purpose computer selectively activated or reconfigured by software, one of ordinary skill in the art would also recognize that
5 such methods may be carried out in hardware, in firmware, or in more specialized apparatus constructed to perform the required method steps.

While particular embodiments of the present invention have been shown and described, it will be obvious to those
10 skilled in the art that, based upon the teachings herein, changes and modifications may be made without departing from this invention and its broader aspects and, therefore, the appended claims are to encompass within their scope all such changes and modifications as are within the true
15 spirit and scope of this invention. Furthermore, it is to be understood that the invention is solely defined by the appended claims. It will be understood by those with skill in the art that if a specific number of an introduced claim element is intended, such intent will be explicitly recited
20 in the claim, and in the absence of such recitation no such limitation is present. For a non-limiting example, as an aid to understanding, the following appended claims contain usage of the introductory phrases "at least one" and "one or more" to introduce claim elements. However, the use of
25 such phrases should not be construed to imply that the introduction of a claim element by the indefinite articles "a" or "an" limits any particular claim containing such introduced claim element to inventions containing only one such element, even when the same claim includes the
30 introductory phrases "one or more" or "at least one" and

indefinite articles such as "a" or "an"; the same holds true for the use in the claims of definite articles.